Security Metrics for Java Bytecode Programs

Bandar Alshammari
College of Computer and Information Sciences
Aljouf University, Saudi Arabia
Email: bmshammeri@ju.edu.sa

Colin Fidge
Science and Engineering Faculty
Queensland University of Technology
Brisbane, Australia
Email: c.fidge@qut.edu.au

Diane Corney
Research Labs, Oracle
Brisbane, Australia
Email: diane.corney@oracle.com

***Abstract*—Although there are many approaches for developing secure programs, they are not necessarily helpful for evaluating the security of a pre-existing program. Software metrics promise an easy way of comparing the relative security of two programs or assessing the security impact of modifications to an existing one. Most studies in this area focus on high level source code but this approach fails to take compiler-specific code generation into account. In this work we describe a set of object-oriented Java bytecode security metrics which are capable of assessing the security of a compiled program from the point of view of potential information flow. These metrics can be used to compare the security of programs or assess the effect of program modifications on security using a tool which we have developed to automatically measure the security of a given Java bytecode program in terms of the accessibility of distinguished 'classified' attributes.**

***Index Terms*—Object-Orientation, Security Metrics, Security Analyser, Java Bytecode**

I. Introduction

Security is a critical aspect of software development, and there exist various approaches which aim to reduce security risks and vulnerabilities either through careful coding practices [1] [2] or through static analysis of the code's properties [3] [4] [5]. However, these techniques require considerable skill and effort to apply successfully, and are not always applicable to pre-existing programs. Alternatively, security metrics which quantify the security level of a given program [6] could offer a 'pushbutton' solution which can be applied easily to given programs.

Most existing security metrics are based on high-level code [6] which does not always give reliable results because analysing source code does not take compiler-specific code generation into account. On the other hand, low-level metrics, e.g., those derived from Java bytecode instructions, could provide better information about the executable program [7] [8].

Dynamic metrics have been studied in many projects due to their importance and reliability [7], including the relationships between static and dynamic coupling metrics [8]. Java dynamic metrics have also been studied extensively, including Dufour et al.'s work [7] which defines a number of Java dynamic metrics to evaluate compiler optimisations. Binder and Hulass' study into control flow metrics is also relevant [9].

In our previous work [10] [11] [12] we defined several security metrics for UML class designs, and described a tool for automatically evaluating such metrics [13]. These metrics assess the potential flow of 'classified' information by measuring the accessibility of selected data items based on the security design principles of "reducing the size of the attack surface" [14] [15] and "granting least privilege" [16] [17] [2]. They allow software developers to easily assess the impact of code modifications on overall program security and to compare the relative security of different versions of the same program.

In this paper, we describe our program code metrics in detail, as an extension of our earlier metrics for UML designs [10] [11] and use a large-scale case study, involving multiple programs, to show how the metrics can accurately measure changes in the security of program code.

II. Program Code Security Metrics Definitions

This section defines our security metrics for assessing the security level of programs. Our approach is based on static analysis of the program code. For instance, when we say that a method 'accesses' or 'interacts with' a classified attribute, it means that the method's compiled code contains an instruction that may read or write an attribute labelled by the programmer as 'classified'. Of course, if this instruction appears within a conditional statement, there is no guarantee that the method will do so every time the program executes. Thus, our metrics are safely conservative and measure the *potential* flow of classified data.

A. Data Encapsulation-Based Security Metrics

Code security metrics based on data encapsulation aim to statically measure the potential flow of information from classified attributes and methods from the perspective of access modifiers. These metrics consider attributes annotated by the programmer as 'classified' or ones which derive their values from classified attributes, and methods which access classified attributes. They also measure whether the program imports the Java reflection library due to the risk that this library can be used by new code to access the value of *any* classified attribute in an existing security-critical part of the program [18].

We divide the metrics for this property into four kinds of accessibility: classified instance attributes (CIDA); classified class attributes (CCDA); methods which access classified attributes (COA) and accessibility through reflection (RPB). The metrics are defined so as to penalise programs that make classified attributes more accessible. Higher values indicate higher accessibility to classified attributes and methods, and

hence a larger 'attack surface'. This means a higher possibility for confidential data to be exposed to unauthorised parties. Aiming for lower values of these metrics adheres to the security principle of reducing the size of the attack surface [14], [18]. Here we define the security metric RPB while the descriptions and definitions of the CIDA, CCDA and COA metrics can be found elsewhere [10].

Reflection Package Boolean (RPB): This code-level metric measures the accessibility through reflection of classified data in a given program. It is defined as *"A boolean value representing whether the Java program imports the Java reflection package (1) or not (0)"*. This metric is only concerned with whether or not the application itself is importing the Java reflection library (i.e., information flow within the program itself) and does not consider an attacker reflecting on the application code elsewhere. The reflection metric equals 1 if the program imports the Java reflection library or 0 otherwise. Importing the Java reflection library means a higher possibility for confidential data to be exposed to unauthorised parties.

$$RPB(P) = \begin{cases} 1, & \textit{if reflection imported} \\ 0, & \textit{otherwise} \end{cases} \quad (1)$$

B. Cohesion-Based Security Metrics

The property of cohesion measures the interactions between attributes and methods within a given class [19]. We have previously defined four cohesion-based security metrics to measure the potential flow of classified information caused by interactions between methods and classified attributes in an object-oriented design [10]. Programs with higher interaction between methods and classified attributes have stronger cohesion, and hence are less secure. The previously-defined metrics are divided into four parts: the interactions of mutators (setters) with classified attributes (CMAI); the interactions of accessors (getters) with classified attributes (CAAI); the weight of classified attributes' interactions with methods (CAIW); and the proportion of classified methods (CMW). In this paper, we extend these design-level metrics to include the proportion of classified writing methods in the program (CWMP).

Classified Writing Methods Proportion (CWMP): This metric aims to measure the proportion of methods which write classified attributes in a particular program. We define this metric as: "*The ratio of the number of methods which write classified attributes to the total number of classified methods in the program*". In our case, we assume a writing method in Java is one which writes an attribute to outside its class by calling a method from the `java.io` package. This includes methods whose class name contains either 'write', 'print', or 'out'. Therefore, a 'classified writer' is a method which uses one of these classes to write a classified attribute. Fewer such methods adheres to the principle of granting least privilege [17].

Consider the set of classified methods in a program P as $CM = \{cm_1, \ldots, cm_n\}$ and the set of the classified writing methods as $CWM = \{cwm_1, \ldots, cwm_n\}$ such that $CWM \subseteq CM$. (Given a set S, let the magnitude operator $|S|$ returns the size of the set.) Then, $CWMP$ is expressed as:

$$CWMP(P) = \frac{|CWM|}{|CM|} \quad (2)$$

C. Coupling-Based Security Metric

Our security coupling metric (CCC) for class designs [11] measures the degree of potential flow of classified data caused by the interactions between classes and classified attributes in a given object-oriented design. This metric is adopted without change for program code.

D. Composition-Based Security Metric

As explained previously [11], composition yields a (weak) possibility of potential information flow for classified data. In the case of programming in Java, it is possible to access composed-part (inner) classes unless they are marked as private. Hence, it is recommended to avoid using non-private inner classes in security-critical code [20]. In our case, we assume that using private composed-part classes should reduce the potential flow of classified data, and hence produce more secure programs. Our design-level composition-based metric (CPCC) [11] is adopted for program code to measure this.

E. Extensibility-Based Security Metrics

To have more secure programs, classes and methods which can access classified data should be prevented from being extended by other classes and methods [20], [21], since doing so makes classified data accessible in the new code. We have identified two metrics (CCE and CME) which reward the use of non-extended classes and methods [11].

Another such threat with regard to this property is code that assigns classified values to a variable or parameter that is not subsequently used, because this makes it possible to add code to the program that accesses the classified data but has no observable effect on the program's behaviour. To prevent this we need to identify classified values that are defined but not used, classified methods that are declared but not called, and critical classes that are never used. Thus, we define three new code-level metrics which penalise unused classified attributes (UACA), uncalled classified methods (UCAM) and unused critical classes (UCAC) in a program.

These features could allow unauthorised parties to acquire privileges on security-critical data without affecting the program's original behaviour. Making code inextensible eliminates this risk, and hence reduces the possibility of information flow from these attributes, methods and classes, which adheres to the principle of granting the least privilege [17].

Unaccessed Assigned Classified Attribute (UACA): This metric is define as "*The ratio of the number of classified attributes that are assigned but never used to the total number of classified attributes in the program*". It measures those classified attributes which are assigned, either directly by an "=" assignment or by parameter passing through value or reference, but never subsequently used.

Consider the set of classified attributes in a program P as $CA = \{ca_1, \ldots, ca_n\}$ and the set of classified attributes which are assigned but never used in the same program as $UCA =$

$\{uca_1, \ldots, uca_n\}$, such that $UCA \subseteq CA$. Then, we define the Unaccessed Assigned Classified Attribute metric as follows.

$$UACA(P) = \frac{|UCA|}{|CA|} \tag{3}$$

Uncalled Classified Accessor Method (UCAM): This metric measures declared methods which access classified attributes but are never called. It is defined as "*The ratio of the number of classified methods that access a classified attribute but are never called by other methods to the total number of classified methods in the program*".

Consider a set of classified methods in a program P as $CM = \{cm_1, \ldots, cm_n\}$ and classified accessors that are never called by other methods $UCM = \{ucm_1, \ldots, ucm_n\}$, such that $UCM \subseteq CM$. Then, we define the Uncalled Classified Accessor Method metric as follows.

$$UCAM(P) = \frac{|UCM|}{|CM|} \tag{4}$$

Unused Critical Accessor Class (UCAC): This measures classes which contain classified accessor methods that are never used in any other classes. It is defined as "*The ratio of the number of classes which contain classified methods that access classified attributes but are never used by other classes to the total number of critical classes in the program*".

Consider the set of critical classes in a program P as $CC = \{cc_1, \ldots, cc_n\}$ and classes which have classified accessors that are never used by other classes as $UCC = \{ucc_1, \ldots, ucc_n\}$, such that $UCC \subseteq CC$. hen, we define the Unused Critical Accessor Class metric as follows.

$$UCAC(P) = \frac{|UCC|}{|CC|} \tag{5}$$

F. Design Size-Based Security Metrics

With respect to security, a program with a large amount of security-critical code has a higher chance of potential flow of classified information, and hence is less secure [6]. It has also been shown that security-sensitive classes must avoid serialisation since this allows the values of private fields to be accessed from outside the program [18]. Our design size-based security metric (CDP) defined previously [11] already measures the proportion of the program that is devoted to security-critical classes (CDP). For program code we define another security metric devoted to security-critical serialisable classes (CSCP).

Critical Serialised Classes Proportion (CSCP): This metric measures the risk associated with critical serialisable classes in a given program. We define it as "*The ratio of the number of critical serialised classes to the total number of critical classes in the program*". It rewards programs with a smaller percentage and number of such classes and penalises the use of security-critical serialisable classes. Therefore, lower values of the CSCP metric indicates a lower proportion of security-critical serialisable classes, which can give privileges over confidential data, and thus satisfies the least privilege principle [17].

Consider the set of the critical classes in program P as $CC = \{cc_1, \ldots, cc_n\}$ and the set of critical serialised classes is $CSC = \{csc_1, \ldots, csc_n\}$, such that $CSC \subseteq CC$. Then, we define the Critical Serialised Classes Proportion metric as follows.

$$CSCP(P) = \frac{|CSC|}{|CC|} \tag{6}$$

G. Inheritance-Based Security Metrics

The earlier design-level metrics [11] which consider inheritance are equally-applicable to program code. The include metrics which penalise classes (CSP and CSI), methods (CMI), and attribute (CAI) hierarchies in which classified data appears near the top and is thus more accessible.

III. Program Code Security Metrics Experimental Results

To demonstrate the validity of our code-level metrics, and show the capabilities of our Java Bytecode Security Analyser [13], we conducted an experiment with several large-scale open source Java programs. We used the tool to assess the relative security of different versions of the same program. Our hypothesis was that a program's level of security should, on average, improve over time, as bugs are fixed and the program code is improved, although the addition of new security-critical functionality may cause a worsening of overall security levels.

A. Approach

We began with five existing open source Java security projects which were chosen from the most frequently downloaded security projects on the SourceForge website [22]. The chosen programs consisted of the following: Jacksum, jGuard, Kasai, JSecurity, and JXplorer. They all mainly concentrate on providing a framework for handling authentication, authorization, enterprise session management, and cryptography services [22]. For each project, we chose a specific version which was modified in a number of subsequent updates, to fix bugs found in the previous releases. In this way we could compare different versions of each program with identical functionality but (hopefully) improved code quality. All of these programs are security-related, so we could reasonably expect successive releases to be more secure than their predecessors.

B. Program Annotations

First, we manually annotated at the Java source code level a number of attributes in each project to be 'classified', choosing attributes whose names and associated code comments indicated that they are likely to store confidential data. We annotated the same attributes for all the different releases of the same program in order to make our comparisons fair. For example, in the program JSecurity, classified attributes were: `username`, `password` and `rememberme` in the `UsernamePasswordToken` class. In the Kasai project, `login`, `password` and `superUser` in the `User` class and in class `Role id` and `name` were

Table I: Program Characteristics

Program	Version	Attributes	Classified Attributes	Methods	Classified Methods	Classes	Critical Classes
JSecurity	0.9.0.A	384	3	1790	24	241	1
	0.9.0.B1	383	3	1822	24	245	1
	0.9.0.RC1	444	3	1996	24	261	1
	0.9.Stable	435	3	2028	23	309	1
Kasai	1.1.0.B1	86	5	498	49	51	2
	1.1.0.B2	86	5	506	54	51	2
	1.1.0.B3	86	5	506	54	51	2
	1.1.Stable	86	5	506	54	51	2
Jacksum	1.2	146	5	179	28	24	2
	1.3	159	7	217	41	29	3
	1.4	260	9	304	44	36	3
	1.5	299	16	355	56	44	3
jGuard	0.65.1	219	4	344	40	45	2
	0.65.2	219	4	344	40	45	2
	0.65.3	219	4	344	40	45	2
	0.65.4	219	4	344	40	45	2
JXplorer	3.2.B1	1985	96	3225	421	406	28
	3.2.B2	2075	113	3332	486	413	28
	3.2.B3	2072	142	3336	558	413	31
	3.2.Stable	2077	151	3345	518	415	31

Table II: Data Encapsulation and Cohesion-Based Security Metrics

Program	Version	CIDA	CCDA	COA	RPB	CMAI	CAAI	CAIW	CMW	CWMP
JSecurity	0.9.0.A	0	0	1	1	0.0228	0.0037	0.0225	0.0134	0
	0.9.0.B1	0	0	1	1	0.0226	0.0036	0.0223	0.0132	0
	0.9.0.RC1	0	0	1	1	0.0208	0.0032	0.0205	0.012	0
	0.9.Stable	0	0	1	1	0.0207	0.0032	0.0206	0.0113	0
Kasai	1.1.0.B1	0	0	1	0	0.031	0.028	0.115	0.09	0
	1.1.0.B2	0	0	1	0	0.033	0.03	0.125	0.11	0
	1.1.0.B3	0	0	1	0	0.033	0.03	0.125	0.11	0
	1.1.Stable	0	0	1	0	0.033	0.03	0.125	0.11	0
Jacksum	1.2	0.8	0	1	0	0.05	0.034	0.08	0.156	0
	1.3	0.6	0	0.976	0	0.047	0.044	0.116	0.189	0.024
	1.4	0.4	0	0.931	0	0.029	0.034	0.088	0.145	0.023
	1.5	0.25	0	0.928	0	0.022	0.03	0.123	0.158	0.036
jGuard	0.65.1	0.5	0	0.85	0	0.026	0.026	0.036	0.116	0
	0.65.2	0.5	0	0.85	0	0.026	0.026	0.036	0.116	0
	0.65.3	0.5	0	0.85	0	0.026	0.026	0.036	0.116	0
	0.65.4	0.5	0	0.85	0	0.026	0.026	0.036	0.116	0
JXplorer	3.2.B1	0.8	0.01	0.94	1	0.0025	0.0028	0.081	0.131	0.007
	3.2.B2	0.8	0.008	0.92	1	0.0025	0.0026	0.088	0.146	0.006
	3.2.B3	0.81	0.02	0.92	1	0.0025	0.0024	0.105	0.167	0.007
	3.2.Stable	0.83	0.019	0.92	1	0.0025	0.0024	0.109	0.155	0.008

annotated as classified attributes. In the Checksum project, `value`, `length`, `separator`, and `filename` in class `AbstractChecksum`, and `val` in class `Crc16` were annotated as classified. In the JGuard project, `name` and `applicationName` in the `JGuardPrincipal` class and `id` and `value` in the `JGuardCredential` class were annoated as classified. In the JXplorer project, `uniqueID` and `addressIP` in the `Name` class and `tag` and `name` in the `ASN1Type` class were annoated as classified.

C. Program Characteristics

Table I shows a number of static characteristics of the studied programs after we annotated our choices of classified attributes for each. The arrows show how each metric has changed since the previous release. Upwards arrows (red) indicate a worsening of security and downwards arrows (green) indicate that security has improved. These characteristics are one of the outputs of the JBSA tool. They include the total number of attributes, classified attributes, methods, classified methods, classes and critical classes for each program. In each successive release of each project, most of these characteristics either grew or stayed the same. For instance, the number of classified methods, i.e., those which may access our annotated attributes, either directly or indirectly, can be seen to grow dramatically in successive revisions of Jacksum and JXplorer.

In order to show that our security metrics reflect the program's true security level, we inspected the code of some of the analysed programs in this experiment. This inspection aimed to show that our security metrics correctly mirror the improvement or worsening of security caused by specific changes to security-relevant code.

For instance, it was found that in program Kasai the second release has added a number of additional methods some of which contain a flow of classified information. One such new method is `overridePassword` in class `User` which interacts with the classified attribute `password` and does similar operations to another existing method `setPassword`. It thus creates an additional access point for classified attributes.

Table III: Coupling, Composition and Extensibility-Based Security Metrics

Program	Version	CCC	CPCC	CCE	CME	UACA	UCAM	UCAC
JSecurity	0.9.0.A	0.0111	1	1	1	0	0.75	0
	0.9.0.B1	0.0109	1	1	1	0	0.75	0
	0.9.0.RC1	0.0103	1	1	1	0	0.75	0
	0.9.Stable	0.0054	1	1	1	0	0.75	0
Kasai	1.1.0.B1	0.036	1	1	1	0	0.38	0
	1.1.0.B2	0.04	1	1	1	0	0.36	0
	1.1.0.B3	0.04	1	1	1	0	0.36	0
	1.1.Stable	0.04	1	1	1	0	0.36	0
Jacksum	1.2	0.087	1	1	1	0	0.71	0
	1.3	0.061	1	1	1	0	0.75	0
	1.4	0.048	1	1	1	0	0.8	0
	1.5	0.033	1	1	0.98	0	0.85	0
jGuard	0.65.1	0.0625	1	1	1	0	0.6	0
	0.65.2	0.0625	1	1	1	0	0.6	0
	0.65.3	0.0625	1	1	1	0	0.6	0
	0.65.4	0.0625	1	1	1	0	0.6	0
JXplorer	3.2.B1	0.0098	1	1	1	0.029	0.74	0.18
	3.2.B2	0.0094	1	1	1	0.032	0.73	0.22
	3.2.B3	0.0086	1	1	1	0.04	0.74	0.23
	3.2.Stable	0.0082	1	1	1	0.037	0.76	0.23

Table IV: Design Size and Inheritance-Based Security Metrics

Program	Version	CDP	CSCP	CSP	CSI	CMI	CAI
JSecurity	0.9.0.A	0.0041	0	0	0	0	0
	0.9.0.B1	0.0041	0	0	0	0	0
	0.9.0.RC1	0.0038	0	0	0	0	0
	0.9.Stable	0.0032	0	0	0	0	0
Kasai	1.1.0.B1	0.039	0	0	0	0	0
	1.1.0.B2	0.039	0	0	0	0	0
	1.1.0.B3	0.039	0	0	0	0	0
	1.1.Stable	0.039	0	0	0	0	0
Jacksum	1.2	0.083	0	0.5	0.174	0.722	0.8
	1.3	0.103	0	0.5	0.232	0.722	0.8
	1.4	0.083	0	0.5	0.2	0.722	0.8
	1.5	0.068	0	0.5	0.221	0.75	0.8
jGuard	0.65.1	0.044	0.5	0	0	0	0
	0.65.2	0.044	0.5	0	0	0	0
	0.65.3	0.044	0.5	0	0	0	0
	0.65.4	0.044	0.5	0	0	0	0
JXplorer	3.2.B1	0.069	0.036	0	0	0	0
	3.2.B2	0.068	0.036	0	0	0	0
	3.2.B3	0.075	0.032	0	0	0	0
	3.2.Stable	0.074	0.032	0	0	0	0

Similarly the second release of the Kasai program overloads an existing security-critical method. Class `KasaiFacade` has two methods called `createUser` that have a flow of classified information. This means that there are more methods in this release which interact with classified information than in the previous release. In fact, these new methods have similar responsibilities as existing ones and could have been avoided.

Another example was found in program Jacksum 1.3 which has a method `getChecksumInstance` that returns classified information and is assigned to a new non-classified attribute `checksum`. The method calling `getChecksumInstance` thus exposes classified information which could be exploited by unauthorised parties. Therefore, we expect the security of Jacksum 1.3 to worsen due to the additional vulnerabilities added to it and our security metrics should reflect this change.

On the other hand, there are cases where a potential vulnerability has been removed from the program in a successive release. This was found in program JSecurity where method `executeLogin` in class `FormAuthenticationFilter` that used to be a potential vulnerability in the third release was deleted from the program's fourth release. This has resulted in reducing the number of insecure methods (i.e., those which interact with security-critical information). Such changes could contribute to improvements in the program's overall security and therefore our security metrics should reflect this improvement.

D. Programs Security Metrics Results

The results of calculating our code-level security metrics (using our JBSA tool) for each release of each project are summarised in Tables II to IV. Given that lower values of each metric are considered more secure, programs whose metrics decrease should be those whose security has improved. We expected these security-related programs would improve their overall security with each new release.

With regard to the results shown in the tables, two of the five programs, JSecurity and Jacksum, show an obvious improvement in their security metrics from previous versions.

An exception is jGuard whose metrics are unchanged for all releases. This suggests that only insignificant changes were made to the code which had no security impact at all. This impression is confirmed by the characteristics in Table I which reveal that no major changes were made to the code's size between releases. (Nevertheless, the program's change log says that a number of small bug fixes were made in each revision.)

The other exception is Kasai whose metrics in Table II have slightly increased in value between releases 1.1.0.B1 and 1.1.0.B2, meaning a worsening in security, after which the program was stable. From Table II this would appear to be because the second release added eight new methods, five of which were 'classified'. These new methods account for the slight increase in three of the cohesion and coupling-security related metrics exhibited by the second version of the program.

A similar case is shown by the results of JXplorer where some of its metrics often increased. The reason for this is clearly shown by the program characteristics in Table I which indicate that the program has had a significant amount of new code added. Thus, the program has major increases in some of its security metrics and worse security overall with regard to those metrics. Nevertheless, some of the program's metrics, including COA, CAAI, CCC and CSCP, have managed to decrease and thus its security has improved in this regard.

Comparing these results with the code inspections described in Section III-C we see that our metrics for these programs have accurately reflected the changes in the security of these programs with regard to either removing or adding potential security vulnerabilities. For instance, in Kasai's second release our security metrics that relate to measuring the security of classified methods have shown that security has worsened in this release as expected due to the addition of new security-critical methods and attributes, which is the case for program Jacksum 1.3 as well. On the other hand, security improved for the JSecurity program's fourth release as a potentially vulnerable method was deleted. This change in the code produced a corresponding decrease in metrics that measure the proportion of classified methods (CMAI and CMW). However because the deleted classified methods also interacted with several *non-classified* attributes, the metric that measures the proportion of classified interactions (CAIW) increased.

From this experiment, we can conclude that our security metrics offer a simple, easy to apply and easy to interpret approach to quantifying the security of a given program, once it is properly annotated.

IV. Conclusion

In this work we have described a number of security metrics for object-oriented programs which are measurable at the level of bytecode instructions. Using this approach we capture the exact behavior of a Java program in the Java Virtual Machine, which gives accurate results. They provide developers with a simple way of identifying and fixing security vulnerabilities which might occur from the perspective of information flow of confidential data. The security metrics were demonstrated using a tool which analyses Java bytecode, applied to actual large-scale Java projects. This case study produced results which match our intuitions about the way a program's security changes as its code is extended or debugged.

References

[1] M. Howard and D. LeBlanc, *Writing Secure Code*. Redmond, Wash.: Microsoft Press, 2002.

[2] G. McGraw, *Software Security: Building Security In*. Upper Saddle River, NJ: Addison-Wesley, 2006.

[3] V. B. Livshits and M. S. Lam, "Finding security vulnerabilities in Java applications with static analysis," in *SSYM'05: Proceedings of the 14th conference on USENIX Security Symposium*, (Berkeley, CA, USA), pp. 18–18, USENIX Association, 2005.

[4] G. Smith, *Principles of Secure Information Flow Analysis*, vol. 27, pp. 291–307. Springer, 2007.

[5] D. Volpano and C. Irvine, "Secure flow typing," *Computers and Security*, vol. 16, no. 2, pp. 137–144, 1997.

[6] I. Chowdhury, B. Chan, and M. Zulkernine, "Security metrics for source code structures," in *Proceedings of the Fourth International Workshop on Software Engineering for Secure Systems*, (Leipzig, Germany), pp. 57–64, ACM, 2008.

[7] B. Dufour, K. Driesen, L. Hendren, and C. Verbrugge, "Dynamic metrics for Java," *SIGPLAN Notices*, vol. 38, no. 11, pp. 149–168, 2003.

[8] Á. Mitchell and J. F. Power, "A study of the influence of coverage on the relationship between static and dynamic coupling metrics," *Science of Computer Programming*, vol. 59, pp. 4–25, January 2006.

[9] W. Binder and J. Hulaas, "Flexible and efficient measurement of dynamic bytecode metrics," in *Proceedings of the 5th International Conference on Generative Programming and Component Engineering (GPCE 2006)*, pp. 171–180, 2006.

[10] B. Alshammari, C. J. Fidge, and D. Corney, "Security metrics for object-oriented class designs," in *Proceedings of the Ninth International Conference on Quality Software (QSIC 2009)*, (Jeju, Korea), pp. 11–20, IEEE, 2009.

[11] B. Alshammari, C. J. Fidge, and D. Corney, "Security metrics for object-oriented designs," in *Proceedings of the Twenty-First Australian Software Engineering Conference (ASWEC 2010), Auckland, 6–9 April* (J. Noble and C. J. Fidge, eds.), (California, USA), pp. 55–64, IEEE Computer Society, 2010.

[12] B. Alshammari, C. Fidge, and D. Corney, "Security assessment of code refactoring rules," in *WIAR'2012; Proceedings of the National Workshop on Information Assurance Research, Riyadh, Saudi Arabia, April 18*, pp. 1–10, VDE, 2012.

[13] B. Alshammari, C. Fidge, and D. Corney, "An automated tool for assessing security-critical designs and programs," in *WIAR'2012; Proceedings of the National Workshop on Information Assurance Research, Riyadh, Saudi Arabia, April 18*, pp. 1–10, VDE, 2012.

[14] M. Howard, "Attack surface: Mitigate security risks by minimizing the code you expose to untrusted users," *Microsoft MSDN Magazine*, vol. 11, 2004.

[15] P. Manadhata and J. Wing, "An attack surface metric," *IEEE Transactions on Software Engineering*, vol. PP, no. 99, p. 1, 2010.

[16] J. H. Saltzer and M. D. Schroeder, "The protection of information in operating systems," in *Proceedings of the IEEE*, vol. 63, pp. 1278–1308, 1975.

[17] M. Bishop, *Computer Security: Art and Science*. Boston: Addison-Wesley, 2003.

[18] Java SE Security, "Secure Coding Guidelines for the Java Programming Language." Sun Developer Network, Version 3.0. http://java.sun.com/security/seccode guide.html [Accessed July 9, 2010].

[19] L. C. Briand, J. W. Daly, and J. K. Wuest, "A unified framework for cohesion measurement," in *Proceedings of the Fourth International Symposium on Software Metrics*, IEEE Computer Society, 1997.

[20] G. McGraw and E. Felten, *Securing Java: Getting Down to Business with Mobile Code*. New York: Wiley Computer Pub., second ed., 1999.

[21] M. Y. Liu and I. Traore, "Empirical relation between coupling and attackability in software systems: a case study on DOS," in *Proceedings of the 2006 Workshop on Programming Languages and Analysis for Security, Ottawa*, (Ottawa, Ontario, Canada), pp. 57–64, ACM, 2006.

[22] SourceForge, "SourceForge source code repository," July 2010. http://sourceforge.net/ [Accessed July 9, 2010].